FASMM: Fast and Accessible Software Migration Method

Louis Forite, Charlotte Hug

FASMM: Fast and Accessible Software Migration Method

Louis Forite, Charlotte Hug
Centre de Recherche en Informatique
Université Paris 1 Panthéon-Sorbonne
Paris, France
louis.forite@gmail.com, Charlotte.Hug@univ-paris1.fr

***Abstract*—With the fast changes of development technologies, organizations often need to migrate their software from a source to a target technology that could comprise a shift in programming paradigm. This operation is not easy and requires precision and structuring. However, in small companies, due to lack of resources (workforce, time, budget…) the migration phase is frequently quickly done and not necessarily in an optimized way: functionalities are not implemented properly, the new architecture is loose and knowledge gained during the migration is not capitalized. This paper presents a method to guide developers in the migration of software functionalities based on model driven engineering techniques and allows capitalizing knowledge as transformation rules, to enable their reuse in future migration projects. This method was built from a case study in a French company that produces software training and support for critical applications.**

Keywords—migration method, intentional process model, model driven engineering, knowledge management

I. INTRODUCTION

With the fast changes of development technologies, organizations often need to migrate their software products from a source to a target technology that could comprise a shift in programming paradigm. In this paper, we studied the case of a French SME that produces training and support software products for critical applications. The company was founded 13 years ago and has already trained more than 400,000 users for major worldwide accounts through its software products. It employs 30 engineers that work on the same site. The development team implements agile best practices as pair programming and daily stand-up. In 2013, the company had to migrate one its flagship software product, an Electronic Performance Support System (EPSS), where the data management was developed in Java/J2EE and the users' interaction part for the website was developed in HTML, JavaScript and Ajax. The server is then developed in Java and the communication with the client is done in Ajax. The final objective of this migration was to ensure the full compatibility of the EPSS with the Oracle E-Business Suite ERP by developing a full Java version of the system.

Several issues can be raised during a migration project and in the case of this French company in particular. We describe them hereafter:

No proper method of migration is defined or followed. To migrate the software product, no method is recommended. The development team defines the structure and border of the functionalities to migrate by consensus. Functionalities evaluated as essential are migrated first (rewriting the code «from scratch» in the targeted technology). There is a poor visibility on the goals and expectations regarding the business: the scope of the functional coverage of the software to migrate is not clearly defined. Nothing guarantees the quality and consistency of the new developments. Rewriting the code "from scratch" is a risky and lax technique considering the fact that there is a programming paradigm shift between the source and the targeted technology which creates additional difficulties.

The software does not have source analysis models, visibility and understanding of the current software is therefore complex and limited. Indeed, in SME, modelling does not seem to be a priority. Little time is spent on models management and methodology. These methods are often time-consuming, or require too specific knowledge. Thus, software models are often set aside in favor of actual developments. The models are nonexistent or not kept up to date. This has a double impact on future developments: the developers have poor visibility of the overall software architecture and developments are less rationalized.

The paradigm shift in programming between the source software and the target software is a difficulty. Each programming paradigm provides a different view of the software and offers different options to implement the product to the developers for a given problem. Software products are coded very differently according to their programming language. There is a real complexity to migrate software products from one programming paradigm to another as it requires to abstract the first to rethink the other. Some mechanisms in a given paradigm can hardly be feasible while in another they can naturally be implemented.

Developers accumulate technical knowledge during the migration project that will eventually get lost. During the software migration, developers involved gain experience in migrating software products from one technology to another; from JavaScript to Java in this case study. This knowledge, although essential during migration, is not capitalized, but only shared, orally or by memos, between developers participating in the project. It could be interesting to capitalize this knowledge to reuse it and share it in future migration projects to earn resources and time.

Several methods have been proposed to migrate software products or legacy systems [23], [24], [25]. However, according to our knowledge, those methods do not take into account:

- The technical difficulty introduced by a shift of programming paradigm between the source and target technology,
- The difficulty of their implementation in SME because of their complexity [24], and time and budget consuming constraints.
- The loss of technical knowledge acquired during the migration as it is not capitalized.

The goal of this paper is to propose a method that allows to simply perform software migrations while capitalizing the accumulated knowledge. The method supports a shift in programming paradigm thanks to model driven engineering concepts. It is aimed at small development teams in SME with standard knowledge in modeling. The proposed approach is called "Fast and Accessible Software Migration Method" (FASMM). We designed it to be easy to learn and to use for any software developers in time and resources constraining context.

We describe FASMM in section 2. Section 3 presents a first evaluation of the method. Section 4 presents the related work and section 5 concludes this paper.

II. FASMM

The purpose of this method is to provide a framework for the functionalities 'migration of a software product while the source and targeted technologies are based on different programming paradigms. This approach has multiple objectives as to:

- Automate certain phases of the migration,
- Ensure that the migrated functionalities fulfill their technical and functional objectives,
- Formalize and capitalize the technical knowledge gained during the project,
- Be simple enough that it can be applied by developers who do not have very advanced knowledge in modeling and metamodeling.

This method is primarily intended for small development teams. However, it requires the involvement of some functional actors for the definition of user goals as it is necessary to ensure that the project will cover the business requirements. Functional actors as business analysts or users are then needed to ensure the success of the migration project. We built FASMM from the experience of a project migration in a French SME.

This section will first present an overview of FASMM, we then describe the artifacts and the process in detail.

A. Overview of the method

The method answers to both problems of migration and knowledge capitalization. The method guides the developers in modeling the functionality to migrate. A system functionality is the ability to perform a set of tasks to achieve a specific goal. The developers can then carry out the migration it-self by applying transformation rules to the existing code and the models. They finally have to validate the migrated functionality. While working on the migration, developers discover transformation rules and add them to the dictionary for knowledge capitalization. The inputs of the method are the application to migrate and the existing related technical (class and state transition diagrams) and functional documents (textual use case, class diagrams). The dictionary of transformation rules is also an input (see section II B.2). The output of the method is the migrated and validated functionality and the updated dictionary of transformation rules. Section II.Q presents a global view of all the inputs and outputs of the method.

Fig. 1 presents the method as a map process model, instance of the Map process metamodel [1]. A map focuses on the different strategies to achieve intentions in a flexible way as each intention is reached independently and when it is achieved with satisfaction, the actor may continue the enactment of the process and achieve new intentions. The nodes represent the intentions and the edges, the strategies to follow to achieve an intention target, from an intention source. A section of a map is a triplet composed of a source intention, a target intention and the corresponding strategy to achieve it. In Fig. 1, the sections in bold are refined as map process models. The method is then presented at different levels of abstraction to ease its understanding and to refine the complex parts of the process to facilitate its use and properly guide developers during the enactment.

We defined five intentions: "get model", "migrate functionality", "validate", "discover transformation rule" and "enrich dictionary". By providing the method as a map, developers will be able to work on the different phases of the migration and at the same time enrich the dictionary by transformation rules: Map allows enacting different paths at the same time according to the reasoning of the developers. The FASMM enactment is then flexible as the developers will be able to follow different strategies to achieve their intentions. For example, to get the models of the functionality to migrate, a developer can enact several times the strategies *by reverse engineering* and *by reviewing existing code* as long as the models are not validated (the intention is not achieved). The whole model should be enacted as many times as there are functionalities to migrate.

In the "branch" covering the migration (lower part of Fig. 1), we identified five strategies to get a model. *By reverse engineering* of the existing code, *by reviewing existing code* and *by reviewing existing models* are complementary as there is often a gap between the models and the code. Therefore, it is necessary to complete the existing models before applying transformation rules. It is possible to get models *by identifying reusable part* of codes and *by identifying unsatisfying parts* of code.

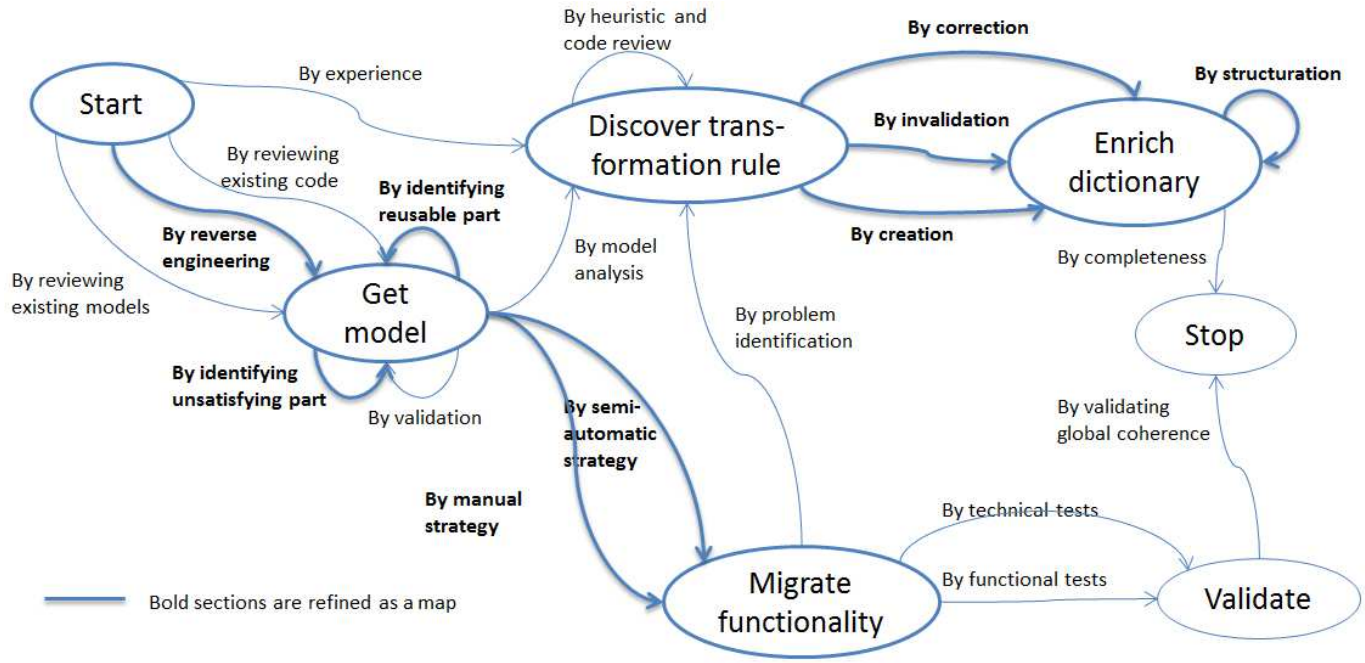

Fig. 1. FASMM presented as a map.

Finally, developers have to validate the technical and functional models by consensus. We identified two strategies to achieve the intention "migrate functionality": *manual* and *semi-automatic*, which are complementary. The manual strategy consists in applying transformation rules from the dictionary (pattern recognition…). The semi-automatic strategy allows generating code from the obtained models. Model Driven Engineering techniques can also be considered to transform models into refined models by using Model to Model transformations. The migrated functionality has to be validated *by functional tests*, i.e. to verify its consistency with the models, and *by technical tests* to ensure the durability of the solution and its reliability.

The second branch of the method is about knowledge capitalization (upper part of Fig. 1). We identified the intentions "discover transformation rule" and "enrich dictionary". Rules can be discovered *by experience*, *by model analysis*, *by problem identification* during the migration of the functionality or *by reviewing the code*. Developers can enrich the dictionary by creating new rule, by invalidating or correcting existing rules or by structuring the dictionary it-self.

The method produces three artifacts: the dictionary that contains the knowledge on migration formalized as transformation rules and the migrated functionality with the associated models. The added value of this method is the knowledge capitalized in the dictionary to be reuse and enrich projects after projects.

We first describe the artifacts produced during the enactment of the method. We then present the sections of the method. Some of them (in bold in Fig. 1, are refined as maps).

B. *The artifacts*

1) *Transformation rules*

"A transformation is the automatic generation of a target model from a source model, according to a transformation definition" and it "is a description of how one or more constructs in the source language can be transformed into one or more constructs in the target language." [2]

We need to specify a transformation rule to identify a specific situation and to propose a solution, for example how to migrate a GUI from Java to JavaScript. To specify a transformation rule, we propose the following template: title of the rule, the source and the target technology, its categories, the application case, its description, its status and the author of the rule.

TABLE I. EXAMPLE OF A TRANSFORMATION RULE

Title	Emulation of an interface in JavaScript
Source Technology	Java
Target technology	JavaScript
Categories	Interface
Application case	An interface can be seen as an abstract class with all its methods as abstracts. As the main purpose of an interface is to force the implementation of the methods it defines, we can simulate an interface by implementing a class with all its methods throwing exceptions. Thus, classes that implement the interface without defining the methods thereof hinder an error.
Description	Java Interface → JavaScript Class (Method → Method throwing exception)*
Status	Active
Author	L.F.

TABLE I. presents an example of a transformation rule to emulate Interfaces in JavaScript from Java.

The proposed template is minimalist; it only comprises six sections to reduce the writing effort of the developers. The experience shows that if the contribution is perceived as a burden, people won't do it [3]. It is then essential to ease the work of the developers. Two sections will then really need an effort: the description and the application case. The description will comprise the transformation itself, described as an algorithm or in a transformation language as ATL [4] or QVT [5].

2) Dictionary

The dictionary is a coherent set of transformation rules on different technologies for the migration. It is organized in categories as HMI, algorithm… The available operations on the dictionary are described in the next sections. The dictionary can be implemented as a Wiki to be easily accessible. The dictionary, once defined, can be reused and enriched in other migration projects (it is then an input of the new migration projects).

3) Migrated functionality and models

The migrated functionality comprises the code and the associated models. At the end of the enactment of the method, the functionality will be technically and functionally tested. It will match the business requirements and technical constraints of upgradability, modularity, maintenance, with a minimal effort of recoding. The migrated functionality will be associated with its updated models. The models are all instances of the UML metamodel [2]: use case diagram for the functional part and class and state-transition diagrams for the technical part of the functionality to migrate. We chose to use those models because they are easy to understand by developers who would not necessarily have strong experience in modelling. The recommended UML models are those which are the most commonly used in companies [28]. Fig. 2 presents a simple technical model defined as a class diagram representing an interface "Payable" and two implemented classes "Invoice" and "Employee" (inspired from [6]). Fig. 3 presents the JavaScript code to represent the technical model using the transformation rule defined in TABLE I.

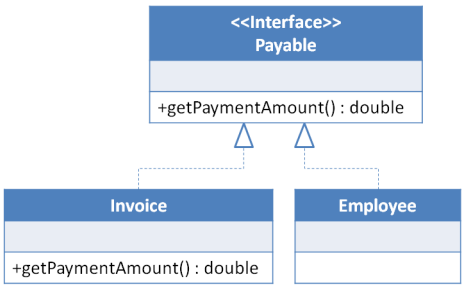

Fig. 2. Simple technical model defined as a class diagram.

To ease the modelling and the transformations, we strongly recommend the use of Integrated Development Environment as EMF [7] for example.

```
(...)
function Payable() { }

Payable.prototype = {
    getPayementAmount: function() {
        throw new Error("The method is not implemented!");
        }
    }

function Invoice() { }

inherit(Invoice.prototype, MonInterface.prototype);  // method to affect the elements from a target class
(superclass) to a source class (subclass)


Invoice.prototype = {
    getPayementAmount: function() {
        alert("Calls the method getPayementAmount");
        }
    }

function Employee() {}

inherit(Employee.prototype, Payable.prototype);

var anInvoice = new Invoice();
anInvoice.method(); // the method is properly executed
var anEmployee = new Employee();
anEmployee.method(); // An error occurs as the method has not been implemented
```

Fig. 3. Code in JavaScript after applying the transformation rule "Emulation of an interface in JavaScript".

C. <Start, Get model, By reverse engineering>

The section <Start, Get model, By reverse engineering> in Fig. 1 is refined in the map presented below (Fig. 4).

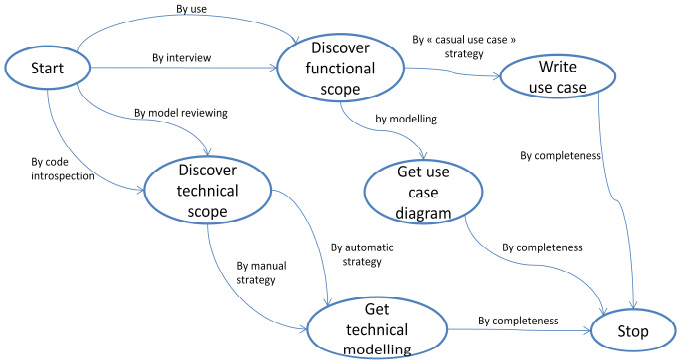

Fig. 4. Refined map of the section <Start, Get model, By reverse engineering>.

There are two different paths in this map: the functional modelling and the technical modelling of the functionality to migrate. The main objective is to understand and define the borders of the functionality to migrate.

1) Functional modelling

Developers have to define a functional scope to better focus on the development of the user goals, using two strategies:

- To interview the people concerned by the functionality to migrate to understand their needs. They can be users or business practitioners. Users are the people who use the system in their daily work. Business practitioners hold the knowledge of the domain of the system, know the related business practices but do not necessarily use the system itself.
- To directly use the functionality in the software to understand its purpose.

These strategies are complementary. Thus, if one of the two strategies is not sufficient to discover the functional scope of the functionality to migrate, the developers may use the other to collect the maximum amount of information.

Once the information gathering is done and satisfying, we recommend the creation of two documents to formalize the functional scope of the functionality to migrate: the textual use case and the use case diagram. The use case diagram should be easily understood, it is in fact a powerful communication notation. In general, the models are an abstraction of reality. They offer a better overview than text documents, but they have the disadvantage of sometimes being less precise. UML [2] is recommended for defining the use case diagrams in FASMM.

On the other hand, the textual use cases allow to describe more precisely a business process as they detail the different steps of a use case. To write a textual use case, the structure defined by Alistair Cockburn, "Casual use case" is recommended [9]. It has the advantage of being concise; however its quality greatly varies according to the author. A textual use case always follows the same template: title, primary actor, scope, level, and story.

Once these two documents are completed, the functional scope is defined. If developers know in advance what the migrated functionality should do, it will be easier to define what part of the code has to be migrated.

2) Technical modelling

Technical models are class diagrams and state-transition diagrams. State transition diagrams can be easily understood by functional actors as it permits to correlate the object of the system and the corresponding business object. We did not preconize the use of sequence diagram that can be hard to define and to understand without good modelling knowledge. To determine the technical models, we identified two strategies:

- *Review existing models* and divide them to determine the borders of the migration. This strategy is strongly recommended when the models match the current architecture of the software as they offer a better comprehension of the system. The developers will then be able to measure the impact of the migration and could tell which class will be useful or not to the migration. However, when models are not up to date, we recommend to base the discovery of the technical scope from the models obtained through code introspection.
- *Review the code by introspection* to understand which part has to be migrated or not when models are not available at all. Although it is tedious and less precise to determine which part of the code to migrate or not, yet, it allows to prepare the developers to the next steps of migration as he/she will need to inspect the code of the functionality.

These strategies can be combined as reviewing the code can help checking the obtained models.

Once the technical scope is defined, the developers can obtain the technical models either by automatically reverse engineering the models from the defined source code (*by automatic strategy),* either by manually designing the models from the reviewed code (*by manual strategy).* Lots of tools exist to reverse -engineer the code as ObjectAid [10], Papyrus [11], eUML2 [12], MaintainJ [13], JS/UML2 [14]… However some technologies are not supported by reverse engineering as JavaScript that is prototyped (dynamic structure during execution). Only libraries written in such languages can be reverse engineered (JQuery [15], script.aculo.us [16]), if the code is written from scratch, it will be impossible to use such tools. The manual strategy is then the only option, although laborious.

D. <Get model, Get model, By identifying reusable part>

The map in Fig. 5 refines the section <Get model, Get model, By identifying reusable part> of Fig. 1.

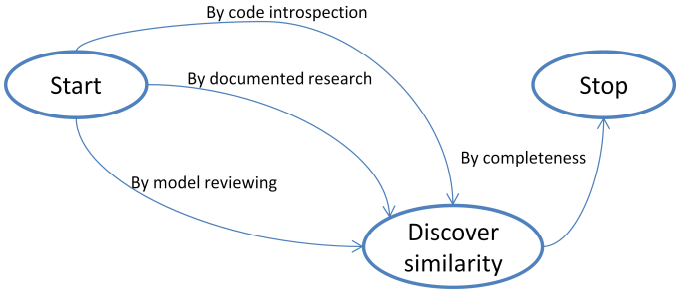

Fig. 5. Refined map of the section <Get model, Get model, By identifying reusable part>.

Before migrating the functionality, we recommend to define what can be reused from the existing code and what should first be transformed. A reusable part is code that is directly exploitable (which can be directly transcribed into the target technology), e.g. a class, a mechanism, a function... This phase is very technical and is addressed to the developers; a lot of time can be spared if this step is done correctly.

To define reusable parts of the software, we defined three strategies:

- *By documented research*,
- *By code introspection* when developers know that the target technology will support this technology or concept (in terms of architecture, design pattern, interface mechanism…).
- *By model reviewing* when the target technology will support the same architecture as the source.

Many commodities [17], [27] exist to make technologies communicate as native API that prevent the developers to recode some functionalities. It is then recommended to do a research to check the possibilities of bridge between source and target technologies and to use them. This will accelerate the development and prevent pitfalls.

For example, in our case study, there is an API provided by Oracle [17] authorizing the invocation of JavaScript code from the applet, and vice versa. This avoids redeveloping functionalities that could cause annoying technical issues such as security problems. In the case of web services, it is easy to reuse WS from a technology to another. It is therefore not necessary to redevelop these services, which will save time and effort.

Many functionalities can actually be reused from the source to the target technologies. This is not the case for all functionalities; typically Graphic User Interfaces are rarely reusable from one technology to another. They must then be redeveloped according to the possibilities of the target technology. Generally, there are similarities between the different technologies for HMI, as listeners or event handlers mechanisms.

With the strategy of code introspection, the developer is asked to dive into the heart of the functionality to migrate to arbitrarily judge what could be subject to direct or indirect reuse.

Finally, reviewing the functional and technical models can also be useful to define what is reusable. Developers will be able to tell whether such architecture is reproducible in the targeted technology given the programming paradigm shift. They should then look specifically into the code if the reusability can be effective.

E. <Get model, Validate, by functional tests>

The validation of the functional modeling is dual: there is firstly a semantic validation that checks the compliance of the use case diagrams [18] [19], and secondly a functional validation to check the clarity of the use cases. A clear use case must be intelligible and understandable by everyone and must fit into the functional scope previously discovered. This validation will be carried out with the technical team (the developers) and the functional team (business analysts) previously involved: are the models relevant enough? Do they precisely reflect what is required in the software in terms of business and user requirements?

F. <Get model, Validate, by technical tests>

The validation of the technical models (class and state-transition diagrams) of the functionality to migrate focuses on the scope of the previously obtained models. These models are the basis of the functionality migration. This validation is therefore done by consensus of the technical team by agreeing on the technical modeling before moving to the migration itself: is the new architecture coherent? Is it scalable, modular and maintainable?

G. <Get model, Get model, by identifying unsatisfying part>

The map in Fig. 6 refines the section <Get model, Get model, by identifying unsatisfying part> of Fig. 1.

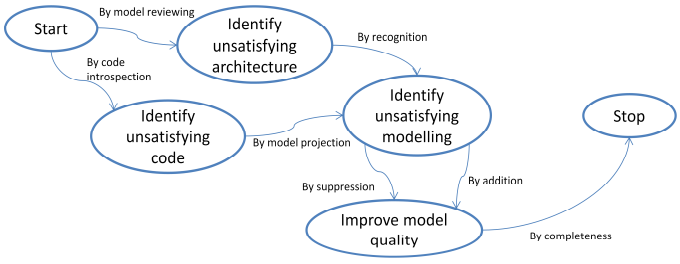

Fig. 6. Refined map of the section <Get model, Get model, by identifying unsatisfying part>.

When developers work with legacy code, technical or architectural choices are questionable. They must then identify which part must be kept or not for the migration and what should be improved. The way to identify non-reusable code must be done within the team by consensus. Non-reusable code does not follow good practices, is unreadable and unintelligible at first glance, contains misnamed classes...Tools allow measuring the quality of code and therefore improve it, as Checkstyle [20] for Java.

The way to identify non-reusable code must be done within the team by consensus.

Once the unsatisfying code is identified, the functionality must be designed in a smarter way. Developers technically model this part of the functionality and integrate it into the global model of the functionality.

H. <Get model, Migrate functionality, by semi-automatic strategy>

The map in Fig. 7 refines the section <Get model, Migrate functionality, by semi-automatic strategy> defined in Fig. 1.

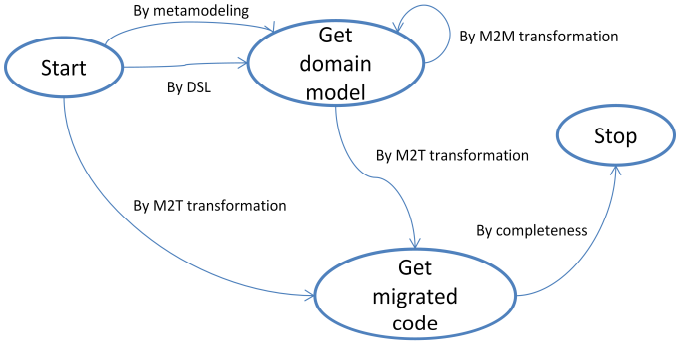

Fig. 7. Refined map of the section <Get model, Migrate functionality, by semi-automatic strategy>.

The semi-automatic strategy is divided into two separate paths. The developers can simply execute Model to Text (M2T) transformations or create complex automated transformations as Model to Model (M2M) and M2T.

M2T transformations allow to automatically generate code from models (*By M2T transformations*). In FASMM, developers generate the skeleton of the future functionality taking as a basis the models obtained by the reverse engineering strategy and complete it with manual transformations rules from the dictionary.

Developers can also go through a more complex model-driven engineering process by defining a metamodel that represents the domain associated to the functionality (*By metamodeling).* Developers must then apply Model to Model (M2M) transformations to refine the models in conformance with the metamodel. This process is very complex to implement and requires advanced knowledge in modeling and metamodeling. We do not recommend this strategy, as most developers lack skills and time to enact it correctly.

However, as previously said, the models obtained by reverse engineering will be partially usable. Some parts of the models will be directly usable (by automatically generating

code) while others will require more subtle and non-automatable transformations.

It is highly recommended to use a maximum of best programming practices to get the best quality code as possible, by writing unit tests for example.

I. <Get model, Migrate functionality, by manual strategy>

The strategy of manual processing is complementary to the strategy of semi-automatic transformation. It consists in applying transformation rules on models or source code. Fig. 8 shows the refinement of the section <Get model, Migrate functionality, by manual strategy> of the method presented in Fig. 1.

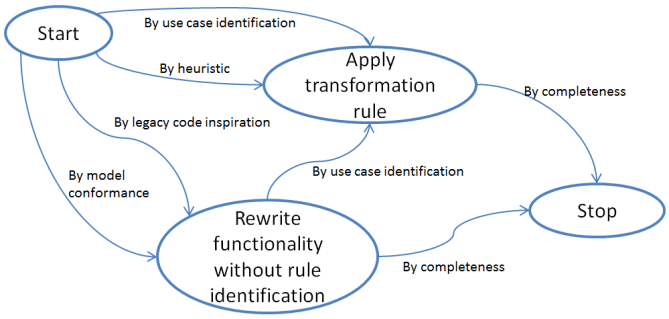

Fig. 8. Refined map of the section <Get model, Migrate functionality, by manual strategy>.

Developers are required to recode manually the functionality. It is initially advised to generate the source code skeleton using a M2T transformation and to manually fill the classes *by model conformance*. Developers have to code what cannot be directly transformed (HMI, architecture incompatibilities, differences between programming paradigms...) *by legacy code inspiration*.

Applying a transformation rule consists first in identifying a particular scenario (in the code, architecture, mechanisms) in which it is necessary to make changes to make it compatible with the target technology. These transformation rules are retrieved from the dictionary that developers complete during the migration. A transformation rule can also be applied *by heuristic* when a developer knows the rule is adapted to the situation.

We also advise to implement a maximum of best programming practices to get a code as good as possible. By writing unit tests for example (if the technology permits it), a smooth development is ensured.

J. Discover transformation rule

We identified four independent strategies to achieve the intention "Discover a transformation rule" (see Fig. 1). The transformation rules will be defined according to the template presented in section II.B.1)

1) By Experience

The strategy of discovery of new transformation rules based on experience enables developers to directly enrich the dictionary when the project starts. Developers accumulated tacit knowledge during previous projects; they can therefore make it explicit by writing transformation rules in the dictionary.

2) By code review / heuristic

During the migration of the software, developers are encouraged to review the code to determine whether some part of the code is reusable. If part of the code is not directly reusable it will require a transformation that will result in a new rule.

The majority of the rules are discovered by code review. Most of the time, these are technical rules to bridge the gap between two programming paradigms.

3) By problem identification

During migration, many technical problems arise: technical incompatibilities, security problems while using API... The identification and resolution of these problems lead to transformation rules as long as the case is identifiable.

4) By model analysis

By analyzing models, it is possible to know which part of the functionality will be reusable or not, the model analysis is then a good way to discover new rules.

K. Enrich dictionary

To achieve the intention "Enrich dictionary", we identified four strategies (see Fig. 1):

- Create a new transformation rule, and add it to the dictionary,
- Disable an existing transformation rule,
- Fix an existing transformation rule by detecting an error, inaccuracy, inconsistency,
- Improve the structure of the dictionary by adding categories (to better find and organize the capitalized knowledge).

In the short term, this dictionary may seem useless because when a developer finds a transformation rule (consciously or unconsciously), he applies it directly. With the dictionary, the developer has to formalize this rule and systematically think about the impact of its results. In addition, the developer is required to write his understanding of the technology. This dictionary can be reused in the following migration projects to earn time while searching for and applying transformations. The dictionary allows to capitalize knowledge in a formalized way.

L. <Discover transformation rule, Enrich dictionary, by creation>

Fig. 9 presents the refinement of the section <Discover transformation rule, Enrich dictionary, by creation> described in Fig. 1.

The creation of a new rule requires its prior discovery and the verification of the existing rules to avoid doubloons. It is then essential for developers to properly identify the rule first by specifying the situation when it applies, its name, its categories, (architecture, algorithm, performance...).

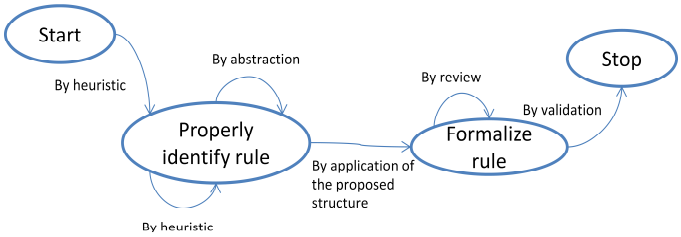

Fig. 9. Refined map of the section <Discover transformation rule, Enrich dictionary, by creation>.

The formalization of a rule by review can be done by other developers to ensure the rule is understandable by everyone.

When several developers reviewed the rule and approved its formalization, the status of the rule can be set to active to be visible by the other team members.

M. <Discover transformation rule, Enrich dictionary, by correction>

The map in Fig. 10 presents the refinement of the section <Discover transformation rule, Enrich dictionary, by correction> specified in Fig. 1.

To achieve the intention "Enrich the dictionary", developers can use the correction strategy. They must first identify the rule; to do so, there are three strategies: the developer can look for a rule *by its name*, *by its category*, or *by keywords*.

Then the developers identify the correction to be made: by adding new content to the rule (when developers discover that the rule can be applied in other cases), by generalizing the case of application of the transformation rule, or by correcting a rule which was initially wrong (if poorly initially specified, if it became obsolete in terms of technology or practice…).

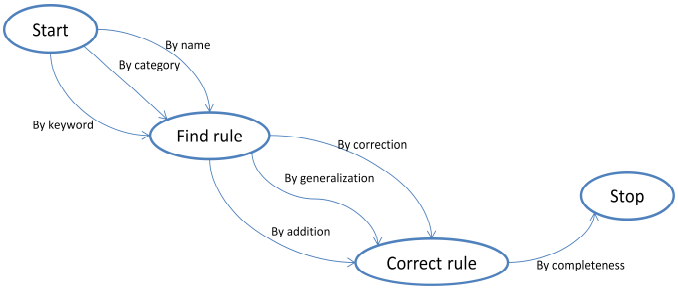

Fig. 10. Refined map of the section <Discover transformation rule, Enrich dictionary, by correction>.

N. <Discover transformation rule, Enrich dictionary, by invalidation>

The map in Fig. 11 presents the refinement of the section <Discover transformation rule, Enrich dictionary, by invalidation> defined in Fig. 1.

The invalidation rule strategy may seem paradoxical to reach the "Enrich dictionary" intention. At first glance, the invalidation is perceived as a loss but removing a false or outdated rule participates indeed in the consolidation of the knowledge of developers. To be useful, the dictionary must contain reliable transformation rules.

It is also possible to archive the rules. This action allows to keep track of their history. This is necessary to understand the existing rules and the process that led to them.

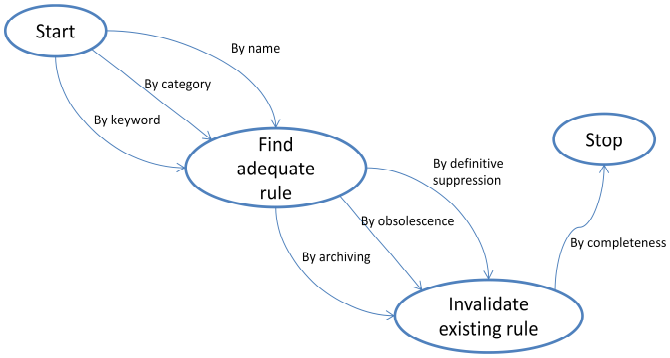

Fig. 11. Refined map of the section <Discover transformation rule, Enrich dictionary, by invalidation>.

O. Validate the migrated functionality

This section is described in Fig. 1. We identified two strategies to validate the migrated functionality. The functionality has to be functionally and technically validated.

1) By functional tests

The functional validation strategy of the migrated functionality consists in checking the conformity of the functionality or part of it to the use cases obtained in the reverse engineering phase. There will be as many tests as use cases.

To validate the migrated functionality it is possible to run end-user tests. If the functionality fulfills its role, it is then validated. Test scenarios can be implemented to ensure the uniformity of functional tests. In addition, questionnaires can be written to enable the collection of users' opinion.

2) By technical tests

The validation by technical tests strategy aims to ensure the technical sustainability of the new architecture and the new code. Developers can verify the implementation of best practices (exception mechanism, use of design patterns, generic code, functionalities developed as components etc.). If there is no legacy code in the new developed functionality, it is then much easier to implement best practices as legacy code often obligates to badly code to keep the system working. It is also possible to test the code by creating unit tests; if they fail the migrated functionality is not validated.

P. <Enrich dictionary, Enrich dictionary, by structuration>

To structure the dictionary, developers can follow three different strategies:

- Create a new category that will better classify the different transformation rules. For example, the HMI elements are not reusable from one technology to another. A developer could create technologic HMI categories that would contain all the knowledge related to specific HMI transformation rules. Another category

could be dedicated to the programming paradigm shift and its implications on the code and how to program from one paradigm to another.

- Modify an existing category to make it more specific or generic depending on the need. For example, the HMI category may be specified into HMI: the forms.
- Classify the rules in different categories to move rules from one category to another.

Fig. 12 shows the refinement of the section <Enrich dictionary, Enrich dictionary, by structuration>.

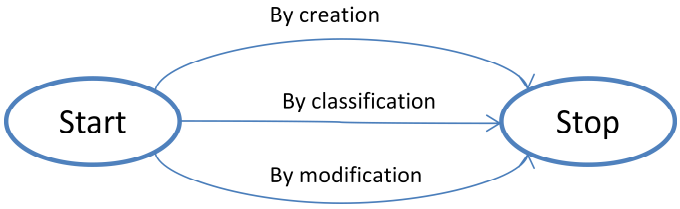

Fig. 12. Refined map of the section <Enrich dictionary, Enrich dictionary, by structuration>.

Q. *Description of the inputs and outputs of the method*

In this section, we focus on the different inputs and outputs of the method. The method is presented as a map; each intention corresponds to the creation or modification of a set of artifacts. We described them in section II.B, however we still need to specify when they are produced and used in the method. TABLE II. presents for each couple of source and target intentions the related inputs and outputs. We do not represent the strategies related to the target intentions as the sections can be enacted as many times as possible until the target intention is achieved. Moreover, different strategies related to the same couple of source and target intentions will be enacted several times to achieve the intention. For instance, from the start intention to the "Get model" intention, the inputs are the existing code, the existing models (functional and technical), and the knowledge hold by the user and the business practitioners on the functionality to migrate. The expected outputs, when the "Get model" intention is achieved, are: the technical models, the textual use cases and the use case diagrams of the functionality to migrate, whatever the applied strategies. We did not detail the inputs and outputs of the refined maps, as it would complicate the understanding of the table.

The final products are presented in the two last rows of the table in bold: the upgraded version of the software or application (each migrated functionality corresponds to an upgrade), and the completed dictionary with the transformation rules discovered during the migration of the functionality.

TABLE II. INPUTS AND OUTPUTS OF THE METHOD

Source Intention	Target intention	Inputs	Outputs
Start	Get model	Existing code, existing models, users and business practitionners knowledge	Technical models, textual use case and use case diagram of the functionality to migrate
Get model	Get model	Technical models, textual use case and use case diagram of the functionality to migrate	Refined and validated models of the functionality to migrate
Get model	Migrate functionality	Refined and validated models of the functionality to migrate Dictionary of transformation rules	Migrated functionality
Migrate functionality	Validate	Migrated functionality	Validated functionality
Start	Discover transformation rule	Developers experience in source and target technology	Transformation rule
Get model	Discover transformation rule	Tehnical models	Transformation rule
Discover transformation rule	Enrich dictionary	Transformation rule Dictionary of tranformation rules	Enriched dictionary of tranformation rules
Enrich dictionary	Enrich dictionary	Dictionary of tranformation rules	Structured dictionary
Validate	Stop	Validated functionality	**Upgraded version of the application**
Enrich dictionary	Stop	Structured dictionary	**Completed dictionary**

III. A FIRST VALIDATION

We set up a protocol to evaluate qualitatively the experience of the developers using FASMM, based on exercises and questionnaires. We carried out the evaluation in the French company, with members of the EPSS migration project.

A. *The protocol*

We conducted the evaluation as semi-structured interviews. First, we gave an overview of FASMM to the subjects, presented its challenges, the whole process as a map, the definition of key concepts as reverse engineering, textual use case, etc... The questionnaire was then distributed with the wording of the exercise to complete. The exercise consisted in following FASMM to migrate a functionality of calculus for an educative software product from HTML5/JavaScript to Java. The code of the functionality in JavaScript and an incomplete class diagram were provided to the subjects. The purpose of the semi-structured interview was to put the subjects in the context of a migration so they could apply different strategies to achieve the intentions following the method. We wanted to evaluate:

- The clarity of the method (its objectives and challenges),
- The ease of use of the method,
- The interest of the developers for the method.

The subjects had to produce different artifacts during the evaluation:

- The artifacts of the method: the models, the migrated functionality, the discovered and formalized transformation rules,
- The products of the evaluation: their comments and remarks.

The actual result and the expected results were compared. We measured the delta between them to evaluate the technical efficiency of the method.

B. *The evaluation and results*

We chose three members of the migration project of the French company as subjects. The evaluation lasted two hours but all the sections of the method were not enacted because of lack of time.

1) *Profile of the subjects*

The three subjects had technical profiles. They all had a good experience in software development (five years minimum), and participated to software migration projects from Java 1.4 to Java 1.5 and Web to Java, without using any predefined method. They then had a good experience in software migration.

2) *Clarity and ease use of the method*

All the subjects agreed the method was understandable with explanations and guidance. Globally, they understood the method and raised an important point: the modelling is important in the migration process.

However, some sections of the method have not been clearly understood as <Get model, Get model, By identifying reusable part> because the "reusable part" concept was misleading: "Is it the code or the architecture?" (S2). Some subjects did not assimilate the concept of transformation rule: "What should be the level of granularity of a rule?" (S1), "When should a rule be used?" (S3).

The subjects felt the method was difficult to use as it was perceived little iterative and directive. The lack of iteration in the method is probably due to the fact the subjects did not understand the principle of Map which allows enacting a section as long as the intention is not achieved. Paradoxically, they did not consider the steps of the method and their implementation too abstract.

3) *Interest for the method*

The subjects manifested a great interest for the method. They were unanimous to say it allowed them to achieve the objective to migrate the functionality proposed in the exercise. They agreed FASMM could be used in an organization according to the context (available time to assimilate the method, specific migration projects).

The subjects regretted the fact of not having enough time to implement some parts of the method, including the functional validation. They were therefore unable to assess this part of the method.

C. *Conclusion*

The subjects all agreed that the exercise was interesting. They underlined that the case presented in the exercise was relevant and that the functionality migration described was justified. The time allotted for the evaluation was too short, however, in two hours, the subjects succeeded in migrating the functionality from JavaScript to Java (without the functional validation).

The results of this evaluation are positive. The method achieved its goal: it guided the subjects through the functionality migration. S1 emphasizes that the migrated functionality was more complete and easier in terms of maintainability, more viable in terms of reusability and genericity than the original functionality. The method helped S2 to overcome the programming paradigm shift thanks to the modelling. S3 described the method as easily affordable.

IV. RELATED WORKS

Performing a software migration consists in transforming an existing system into a new one in a new environment, without redeveloping everything from scratch, to meet the requirements that the old system can no longer ensure [22]. These requirements may be of different nature: new business requirements, performance, maintainability, safety...

The challenges of a migration are multiple. Paradoxically, few software migration methods have been proposed:

- The "Big Bang" or "cold-turkey" method is not actually a method and it is risked. The code is redeveloped from scratch using recent technologies. It is hard to redevelop a software product without reproducing errors present in the legacy code. The knowledge about the software product influences the recoding. There is a high risk of failure [23].
- The "Forward migration" method [23] focuses on the data. Incrementally, developers first migrate the data, then the software product, and the HMIs. Gateways are developed to link the old software product to the data in the new environment. However, gateways can be difficult to handle and make the migration more complex as the number of gateways increases.
- "Reverse migration" method [23] focuses on the software product. The migration is done gradually while data stays in the previous environment. Data migration is the last step of the method. Gateways (reversed) are also implemented between the new and the old environment.
- The "Chicken Little Methodology" [23] is incremental and iterative, the software products evolve continuously. Three types of software products are distinguished: those with a decomposable structure where the interface, the application and the data are independent components. The semi-decomposable structure comprises application and data that are not independent, and then more difficult to migrate. Finally, non-decomposable structures do not provide any independent components. Many gateways are defined between the old and new environments which make the method complex and hard to enact technically.

- The "Butterfly" method [24] was proposed to avoid the gateways problem in a six steps method: prepare migration, understand legacy code, prepare data migration, incrementally migrate the components, incrementally migrate data and finalize new system. Each phase is independent and decomposed in subtasks.
- The SOMA (Service-Oriented Modeling and Architecture) method [25] allows migrating legacy systems to a SOA environment. SOA architecture is made for large systems, the SOMA method is then quite heavy to implement for SME.

These methods are very specific and difficult to implement in SME as they require budget and the participation of business experts as technical experts. Some of them are complex to handle and require specific knowledge that most developers do not hold. Moreover, once the migration is completed, part the knowledge disappears as it is not capitalized during the migration project itself: none of the method is concerned with knowledge capitalization. Finally, the paradigm shift issue is often left aside [26] although it is a complex problem.

The proposed approach in this paper allows precisely to capture and organize this knowledge through a dictionary as transformation rules. Thus, a company with a certain technology and wishing to migrate its software products to a new technology will gain efficiency on each new migration. Finally, FASMM aims to be accessible to any team of developers with basic knowledge in the field of modeling.

V. CONCLUSION AND PERSPECTIVES

Actual migration methods are not adapted to SME because they lack resources to learn and apply the existing software migration methods properly. FASMM was developed in the framework of a migration project in a French company to guide developers during a software product migration, based on the concepts of Model Driven Engineering and knowledge capitalization: models are defined and transformed to get the migrated functionality, using transformation rules stored in a dictionary that can be reused and enriched projects after projects. The method was evaluated with a couple of subjects and results are promising.

Processes in FASMM are specified as a map [1] which allows flexibility in its enactment, as the developers can carry out functionality migration and knowledge capitalization at the same time. Some sections are refined as maps themselves; developers can then follow the process according to their knowledge, ways of working or needs. Being based on Model Driven Engineering techniques, FASMM allows developers to represent the functionalities to migrate as functional and technical models. This facilitates their understanding of the functionality and therefore eases the migration itself. Developers validate the migrated functionality and the produced models represent the code in an efficient and complete way.

There are still many challenges to tackle to complete the proposed method for software migration. First, FASMM should consider the migration of the data which is a problem of software migration. Functionalities are migrated one by one but the method has to propose how to deal with the data between the source and the target software product: is the data migrated first? Does the structure of the data also changes? So does the database technology?

FASMM should be improved to properly take into account the paradigm shift. There are migrations within the same programming paradigm, Java to JavaScript in the Object Oriented paradigm for example. Other paradigms as imperative, functional or logical programming should be supported by the method. As FASMM is based on Object Oriented modelling, we then raise the following question: what would be the best fitted metamodels to support the modelling of the functionalities and their transformations? New types of artifacts have to be introduced to accurately support the migration.

We have to test and validate the method in other software migration projects. It is necessary to evaluate the understanding of the method, its ease of use, and then, to enhance it according to the obtained feed-back.

Rules that can be applied automatically have to be systematically described in a transformation language (ATL [4], QVT [5] or another, according to the knowledge of the developer) to accelerate the migration of the functionalities and to ensure the quality of the migrated functionality.

Finally, we could also represent FASSM as method components to describe precisely the process of each section of the map and the corresponding products to ease the guidance of developers –as one of the answers to the evaluation. Adopting a Situational Method Engineering approach, we could build a component base to store FASMM components, to improve the knowledge on migration methods by adding new components that could be alternative or complementary to FASMM.

REFERENCES

[1] C. Rolland, N. Prakash, and A. Benjamen, "A Multi-Model View of Process Modelling," Requirements Engineering, Vol.4, N. 4, Springer-Verlag London Ltd, 1999, pp. 169-187.

[2] W.J. Kleppe, and W. Bast., MDA Explained, The Model-Driven Architecture: Practice and Promise. Addison Wesley, 2003.

[3] I. Rus, and M. Lindvall, "Guest Editors' Introduction: Knowledge Management in Software Engineering," Vol. 19, N. 3, IEEE Software, 2002, pp. 26-38.

[4] Atlas Transformation Language, https://www.eclipse.org/atl/, consulted in February 2014.

[5] OMG, Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, Version 1.1, 2011.

[6] Paul J. Deitel, and H. M. Deitel, Java for Programmers. Deitel Developer Series, Prentice Hall Professional, 2009.

[7] Eclipse Modeling Framework, https://www.eclipse.org/modeling/emf/, consulted in February 2014.

[8] OMG, Unified Modeling Language, Superstructure, Version 2.4.1, 2011.

[9] A. Cockburn; Writing Effective Use Cases, 2000, Addison Wesley.

[10] Object Aid, http://www.objectaid.com/, consulted in February 2014

[11] Papyrus, http://www.eclipse.org/papyrus/, consulted in February 2014

[12] Soyatec, http://www.soyatec.com/euml2/, consulted in February 2014

[13] MaintainJ, http://maintainj.com/, consulted in February 2014

[14] JsUML, http://jsuml.gaertner-network.de/, consulted in February 2014

[15] jQuery, http://jquery.com/, consulted in February 2014

[16] ScriptAculous, http://script.aculo.us/, consulted in February 2014

[17] Oracle, Invoking JavaScript from applet, http://docs.oracle.com/javase/tutorial/deployment/applet/invokingJavaScriptFromApplet.html

[18] B. Charroux, and A. Osmani, UML 2 3rd edition, 2010, Pearson.

[19] E. Gottesdiener, Use Cases: Best Practices, 2003, IBM.

[20] Checkstyle 5.7, http://checkstyle.sourceforge.net/, consulted in February 2014

[21] M.-M. Saarelainen, J. J. Ahonen, H. Lintinen, J. Koskinen, I. Kankaanpää, H. Sivula, P. Juutilainen, and T. Tilus, "Software modernization and replacement decision making in industry: a qualitative study," Proceedings of the 10th international conference on Evaluation and Assessment in Software Engineering, B. Kitchenham, P. Brereton, M. Turner, S. Charters (Eds.). British Computer Society, Swinton, UK, 12-21, 2006.

[22] J. Koskinen, H. Lintinen, H. Sivula, and T. Tilus, "Evaluation of Software Modernization Estimation Methods Using NIMSAD Meta Framework," Information Technology Research Institute, 2004.

[23] M. L. Brodie, and S. A. Stonebraker, DARWIN: On the Incremental Migration of Legacy Information Systems, 1993.

[24] W. Bing, D Lawless, J. Bisbal, R. Richardson, J. Grimson, V. Wade, and D. O'Sullivan, "The Butterfly Methodology: a gateway-free approach for migrating legacy information systems," Proceedings of the 3rd IEEE International Conference on Engineering of Complex Computer Systems, pp. 200-205, 1997.

[25] A. Arsanjani, S. Ghosh, A. Allam, T. Abdollah, S. Gariapathy, and K. Holley, "SOMA: a method for developing service-oriented solutions," IBM Systems Journal, vol. 47 (3), pp. 377-396, July 2008.

[26] O. Pastor, and J.C. Molina, Model-Driven Architecture in Practice: A Software Production Environment Based on Conceptual Modeling. Springer-Verlag, New York, Secaucus, NJ, USA, 2007.

[27] Oracle, Invoking Applet Methods From JavaScript Code , http://docs.oracle.com/javase/tutorial/deployment/applet/invokingAppletMethodsFromJavaScript.html

[28] B. Dobing and J. Parsons. "How UML is used," Commun. ACM, vol. 49 (5), pp. 109-113, May 2006.